Using Streaming SIMD Extensions 2 (SSE2) for Single-Point Crossovers in Binary Genetic Algorithms

Version 2.0

7/00

Order Number: 248601-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Updated to use the approved terminology of the Pentium® 4 processor	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. Randy L. Haupt and Sue Ellen Haupt, *Practical Genetic Algorithms,* John Wiley & Sons, Inc., New York, New York, 1998
2. Lefteri H. Tsoukalas and Robert E. Uhrig, *Fuzzy and Neural Approaches in Engineering,* John Wiley & Sons, Inc., New York, New York, 1997
3. Eric D. Evans, *Customized Hash Functions Through Genetic Programming,* Purdue University Masters Thesis, 1995
4. Stephen T. Welstead, *Neural Network and Fuzzy Logic Applications in C/C++,* John Wiley & Sons, Inc., New York, New York, 1994
5. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1991

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note contains both the code and a description of how SSE2 can be used to perform a single-point crossover operation used in a binary genetic algorithm.

2 Genetic Algorithms

Genetic algorithms were first developed by John Holland in the early 1970s and form the basis of most evolutionary computation. The idea behind evolutionary computation is to solve a problem by "evolving" a solution. This is accomplished by forming a pool of solutions of which only the best solutions are kept (survival of the fittest). The solutions are then combined (mated) to form new solutions. This new generation of solutions is then evaluated and the process continues until a solution is found that meets a predefined criterion. Binary genetic algorithms form a significant subset of evolutionary computation. This application note deals specifically with one operation that is performed within a binary genetic algorithm, the single-point crossover operation.

There are nine basic components of a binary genetic algorithm (Haupt, 1998):

1. Define Parameters, Cost Function, Cost
2. Represent Parameter
3. Create Population
4. Evaluate Cost
5. Select Mate
6. Reproduce
7. Mutate
8. Test for Convergence
9. Stop if Convergence, Return to Evaluate Cost (4) if no Convergence

Generally speaking, optimization algorithms such as genetic algorithms are presented in such a way that they minimize a particular cost function. One common example might be the Traveling Salesman problem, where the salesman wishes to know the shortest route (minimal distance) to visit N cities.

The parameters of a cost function are called "genes." Genetic algorithms generally represent genes in a binary manner. The concatenation of the binary representation of a set of genes makes up something called a "chromosome." A chromosome is actually a solution to the problem. In the Traveling Salesman analogy, a chromosome may consist of a string of X, Y coordinates of the N cities that form that solution.

A chromosome's cost is called its "fitness." In the Traveling Salesman analogy, the cost or fitness is directly related to the distance traveled, since this is the cost that we are trying to minimize. Each chromosome is evaluated and only the fittest chromosomes (or shortest distance paths) are kept from generation to generation. These chromosomes are allowed to produce new chromosomes through a process called mating.

Mating or crossover involves the exchange of binary code between two chromosomes to produce two new chromosomes. In the Traveling Salesman analogy, mating (crossover) will create new strings of X, Y coordinates of N cities, forming new solutions. The mutation operation randomly alters a few bits in the population, also randomly generating new solutions.

The minimal or lowest possible cost (the fittest chromosome) is the optimal solution. Computing the optimal solution however, may take too long. Consequently, computational tradeoffs are made. A minimum acceptable cost (or fitness) is defined. When a member of the population meets the minimum acceptable cost, the solution is accepted. In the Traveling Salesman analogy, a pre-defined acceptable distance is compared to our shortest distance paths. When one of the paths is equal to or less than the minimum acceptable distance, we accept the path as our solution.

In summary, genetic algorithms provide a powerful computational technique for solving problems that are analogous to evolution. The next section lists some real world problems that have been solved using genetic algorithms.

2.1 Use of Genetic Algorithms

Financial Forecasting

Wall Street has long been aware of the power of genetic algorithms. Numerous companies provide genetic algorithm based financial forecasting software. Most of this software is highly proprietary and the algorithms themselves are well kept secrets.

Scheduling

Scheduling problems are naturally optimization problems, since the problem itself is NP-Complete (this means that this class of problems is not solvable in a satisfactory amount of time. For a further description of this class of problems, please see Cormen, Leiserson, and Rivest, 1995). Genetic algorithms have been very successful in finding good schedules in an acceptable amount of time. Consequently, there are a number of companies offering genetic algorithm based scheduling software for factory, machine and network scheduling.

Radar and Communications

From stealth design to the simple antenna, genetic algorithms have been used to solve numerous radar and communications problems (Haupt, 1998).

Astrophysics

Genetic algorithms have been used for searching for planets around pulsars, "modeling the rotation curves of galaxies, extracting pulsation periods of Doppler velocities in spectral lines, and optimizing a model of hydrodynamic wind" (Haupt, 1998).

Signal Processing

Genetic algorithms are extensively used in signal processing "to adapt isobutylene-isoprene rubber (IIPR) filter coefficients," attenuation of noise, and speech processing (Haupt, 1998).

Geophysics

Genetic algorithms have been used for determining the type of underground rock layers, locating the epicenter of an earthquake, determining the source of air pollutants, and for oceanographic experimental designs (Haupt, 1998).

Other

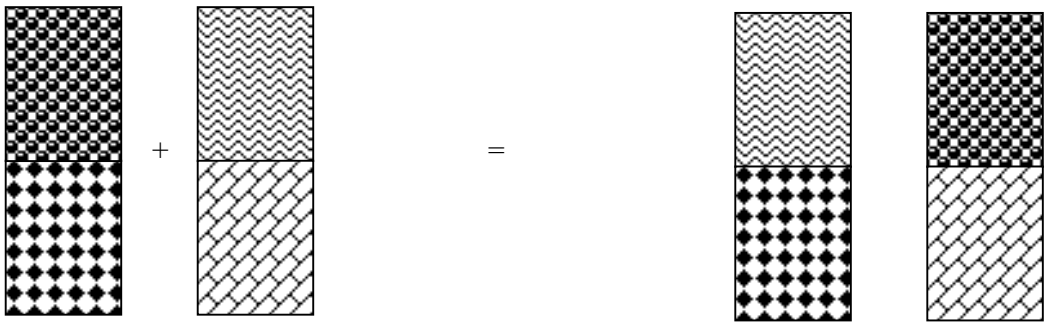
While this is not an exhaustive list, it should be noted that there is a large class of optimization problems that naturally lend themselves to being solved by a genetic algorithm.

"In the genetic algorithm literature, parameter interaction is called epistasis (a biological term for gene interaction). When there is little to no epistasis, minimum seeking algorithms perform best. Genetic algorithms shine when the epistasis is medium to high, and pure random search algorithms are champions when epistasis is very high" (Haupt, 1998).

In general, whenever you are optimizing over a nonlinear multimodal cost surface, genetic algorithms are proven to be superior problem solvers.

2.2 Implementing the Single-Point Crossover Operation

Mating or crossover in a binary genetic algorithm is most simply accomplished around a single point. In our example, we represent each chromosome by using a 16-bit integer. Our crossover point is in the middle. Consequently, the lower 8 bits of parent one are combined with the upper 8 bits of parent two to produce one offspring. The upper 8 bits of parent one are combined with the lower 8 bits of parent two to produce the second offspring.



2.2.1 Techniques

Typically, a bit mask is created which masks off the unwanted bits from each of the parents that we wish to combine. An example of setting a bit mask in C follows:

```
unsigned short int mask1 = 0xff;
unsigned short int mask2 = 0xff00;
```

Once the bit mask is created, it is logically ANDed with the binary string of the parent to mask off unwanted bits. The results of this operation are logically ORed together to obtain the resulting child.

Child1 = ((Parent1 & mask1) | (Parent2 & mask2));

This application note is specifically oriented towards a single-point crossover operation. The techniques discussed and illustrated, however, may also be used with alternative crossover operations by altering the bit mask used in the operation.

SSE2 instructions allow us to more efficiently load and operate on the parents in a crossover operation. Specifically, 8 sixteen-bit integers can be loaded at one time. These integers can be simultaneously operated on using SSE2. This allows us to mate 8 pairs of parents at a time. Using the SSE2 C++ Double Precision Vector Class (DVEC) instruction set, the actual code is virtually identical to the C version. The only difference is in the data types.

2.2.2 Tips and Tricks

1. This application note assumes that the fronts of the parent and child vectors are each aligned on a 16-byte memory boundary. This is a requirement of the SSE2 DVEC class. Users who want to use unaligned data should use SSE2 Intrinsics. A significant performance penalty is incurred if you use unaligned data.

   The Intel® C/C++ Compiler provides you with the following compiler directive for insuring data alignment:

   ```
   __declspec(align(16)) unsigned short int parents[NUMBER];
   ```

   Check your compiler documentation for ways to insure the proper alignment.

2. The C/C++ shift operator can be used to multiply or divide numbers by a power of 2. Rather than calculating indexes in this example by division, shifts are used to improve performance. For example, a division by 8 can be expressed as a shift right by 3.

3. The order of the resulting children is different in the C-code Version versus the SSE2 version. This is resulting from the fact that 8 pairs of parents are loaded and operated on simultaneously in the SSE2 Version, rather than operating on one pair at a time as is done in the C-code Version. The resulting children however, are the same.

3 Performance

3.1 Gains/Improvements

The class code implementation using SSE2 is significantly faster than its corresponding C implementation. This was due to the use of SSE2 that allows us to load and operate on 8 sixteen-bit elements at a time. These instructions are significantly faster than x86 instructions, which can only operate on a single sixteen-bit element at a time.

3.2 Considerations

Larger population sizes in genetic algorithms usually result in better solutions to the problem. In this example, SSE2 allow us to operate on 8 times the number of chromosomes that we could before. Readers employing genetic algorithms should consider scaling up their population to take full advantage of these new instructions.

4 Conclusion

SSE2 can significantly improve the performance of genetic algorithms. While this application note has focused on a single-point crossover operation, other operations such as calculating the fitness of a chromosome may experience significant improvements in performance as well.

5 C/C++ Coding Example

```
void cross_c(unsigned short int *parents1, unsigned short int *parents2,
unsigned short int *children, int number)

{

   unsigned short int mask1 = 0xff;

   unsigned short int mask2 = 0xff00;

   for(int i=0; i<number; i++) {

      children[2*i] = ((parents1[i] & mask1) | (parents2[i] & mask2));

      children[2*i+1] = ((parents1[i] & mask2) | (parents2[i] & mask1));


   }

}
```

6 SSE2 DVEC Code Example

```
void cross_i(unsigned short int *parents1, unsigned short int *parents2,
unsigned short int *children, int number)
{
  // Check for alignment.  The vectors must be 16-byte aligned.
  if(
  (((unsigned int)&parents1[0] & (0x0F)) != 0) ||
  (((unsigned int)&parents2[0] & (0x0F)) != 0) ||
  (((unsigned int)&children[0] & (0x0F)) != 0)
  ) exit(1);  // Not 16 byte aligned

  // Set our masks
  Iu16vec8 p1_mask(0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff);
  Iu16vec8 p2_mask(0xff00,0xff00,0xff00,0xff00
                ,0xff00,0xff00,0xff00,0xff00);

  // Declare pointers to the vectors of our class type
  // This SIMD class type is composed of 8, 16 bit unsigned integers
  Iu16vec8  *p1 = (Iu16vec8 *)parents1;
  Iu16vec8  *p2 = (Iu16vec8 *)parents2;
  Iu16vec8  *ch = (Iu16vec8 *)children;

  // Divide the number of parents by 8 and assign the value to big_bites
  // big_bites is the number of times we need to load/operate on parents
  int big_bites = number>>3;
  for(int i=0; i<big_bites; i++) {
    ch[2*i] = (p1[i] & p1_mask) | (p2[i] & p2_mask);
    ch[2*i+1] = (p1[i] & p2_mask) | (p2[i] & p1_mask);
  }

  // k is the number of parents that have been completed in the loop above
  // j is the index at which we start adding children
  // Finish off the balance with C code
  unsigned short int mask1 = 0xff;
  unsigned short int mask2 = 0xff00;
  int j = big_bites << 4;
  int k = big_bites << 3;
  for(i=k; i<number; i++) {
    children[j] = ((parents1[i] & mask1) | (parents2[i] & mask2));
```

```
      children[j+1] = ((parents1[i] & mask2) | (parents2[i] & mask1));
      j+=2;
   }

}
```

Appendix A – Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium® 4 processor performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of Genetic Algorithm Implementations

Performance Data in Microseconds		
Cases	**Pentium III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
C Code	17.4	11.3
Streaming SIMD Extensions 2 (SSE2) DVEC	-	2.17

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 Processor (SSE2 DVEC vs. C Code)	5.20
C Code on Pentium 4 Processor vs. C Code on Pentium III Processor	1.54

Table 1 and 2 measure the performance of a 1.2 GHz Pentium 4 processor and a 733 MHz Intel® Pentium III processor when executing the genetic algorithm code. See Test System Configurations for a detailed description of the Pentium 4 and Pentium III processor systems. Performance on both processors was measured with test data in the level one cache.

A vector length of 1000 elements was chosen for representation. Full compiler optimizations were used with the Intel® C/C++ Compiler to compile the C code. Significant performance improvements were gained by the use of SSE2 that allowed operations on 8 16-bit integers at a time. The speedup is not 8x owing to resource constraints.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195